LA-UR-21-27508
Approved for public release; distribution is unlimited.

Title: Computing Angular Distributions from Simulation Data

Author(s): Woods, Charles Nathan

Intended for: Report

Issued: 2021-07-29

Computing Angular Distributions from Simulation Data

C. Nathan Woods, XCP-8

Introduction

Overview

The essential idea of this algorithm is to compute the angular distribution of a vector quantity, then create an informative image. In our example, we will compute the angular distribution of linear momentum from an xRage simulation of an exploding shaped charge. We will then explore one possible method for adding information to the resulting image.

Figure 1 - Axisymmetric slice of exploding shaped charge, showing the different materials involved. The exploded primer charge is on the left, the shock wave separating the reactants and products in the main charge is located in the center. The thin layer of metal can be seen forming a jet toward the right, and the thick metal casing surrounds the charge on the top. The axis of symmetry is at the bottom.

Computing Polar Angle

The first thing to consider is how to properly define and compute the polar angle. In our case, we have a two dimensional simulation with an axis of symmetry oriented along the x-axis. We will measure the polar angle from the positive x-axis. Paraview does not offer an arctangent function that gives correct quadrants, so we will define one using a Programmable Filter, as shown in Programmable Filter 1.

```
p = inputs[0].CellData['cell_momentum']

def atan2(y,x):
  sgnx = sign(x)
  sgny = sign(y)
  # This fails when x = 0, b/c arctan(y/x) = NaN, and 0*NaN = NaN.
  # Make sure you've gotten rid of those using Threshold or similar.
  out = sgnx**2*arctan(y/x)+.5*(1-sgnx)*(1+sgny-sgny**2)*pi
  return out

output.CellData.append(atan2(p[:,1], p[:,0]), 'theta')
```

Programmable Filter 1 - Code for computing the polar angle based on (x,y) components, as measured from the positive x-axis.

As noted in the code, the arctan formula will fail for inputs with no momentum in the x-direction. This is most easily managed by first using a Paraview Threshold filter to remove all data with x-momentum smaller than some epsilon value.

Computing Angular Distribution of Momentum

Mathematically, we wish to compute the distribution of momentum in the simulation as a function of polar angle. This is most conveniently done mathematically by expressing momentum in polar coordinates (p, θ) and integrating over the magnitude p. Unfortunately, in our case we do not have access to a continuous field representation of momentum, so we must modify our algorithm slightly.

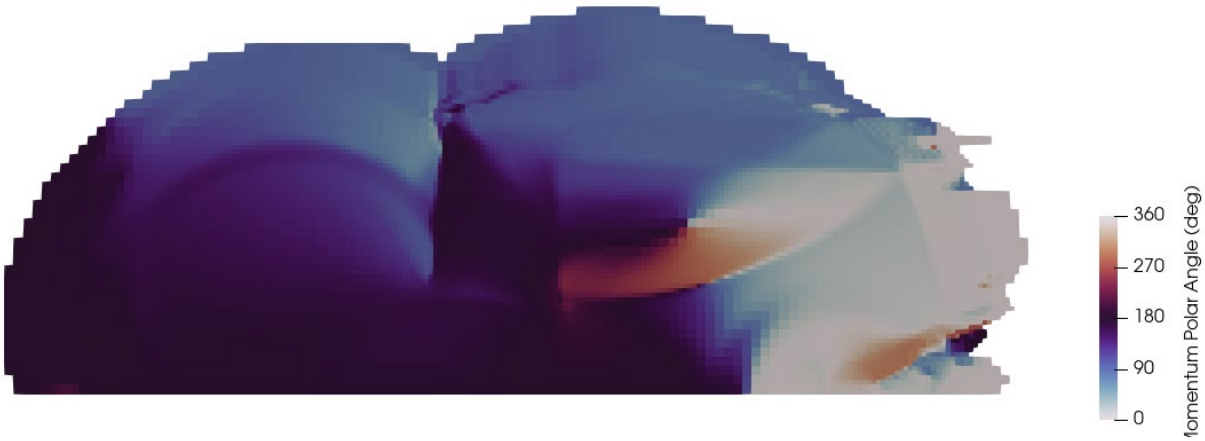

Figure 2 - The polar angle of the momentum vector as a function of space. Note the reddish areas which are moving radially inward (up), while the blue of moving radially out (down). Areas with zero momentum (such as the unreacted main charge) are not shown.

Suppose that we have a field $\phi: M \to \mathbb{R}^n$ and we wish to integrate it over some surface $S \subset M$ (such as a cone of constant polar angle). The field ϕ is represented as a set of scalar or vector values at discrete points (denoted by ϕ_i) which do not necessarily coincide with the surface S. Using continuous functions and variables, we would compute the integral of ϕ over S directly:

$$\Phi = \iint_S \ \phi \, dS.$$

Since we do not assume that we have a set of discrete facets that aligns with the grid, we must find some way to approximate this integral. We do not choose to interpolate grid values onto the aligned surface. Instead, we will perform a weighted integration, with the weight function chosen to include only those values that ought to affect the average. In choosing our weight function, we want something that varies smoothly across its extent while ignoring points that are far from the averaging surface. One example is the bump function:

$$\Psi(x) = \begin{cases} \exp\left(\frac{1}{(x^2-1)}\right) & ; \quad x \in (-1, 1) \\ 0 & ; \quad otherwise \end{cases}$$

In our case, we want to be able to control the width of our bump, which yields:

$$\Psi(x,h) = \begin{cases} \exp\left(\frac{h^2}{(x^2-h^2)}\right) & ; \quad x \in (-h,h) \\ 0 & ; \quad otherwise \end{cases}$$

We will need to normalize this function for a given value of h:

$$\Psi'(x,h) \equiv \frac{\Psi(x,h)}{\int_{-h}^{h} \Psi(x,h)dx} \cong \frac{\Psi(x,h)}{0.443994\ h}$$

It is straightforward to show:

$$\lim_{h\to 0} \iiint_V \Psi'(x,h)\phi\big(V(x)\big)dV = \iint_S \phi(S)dS$$

From this, we may define:

$$\iint_S \phi\ dS \approx \sum_i \Psi(x_i,h)\ \phi_i\ V_i$$

Evaluating the surface integral in this way is mathematically equivalent to performing a smoothing convolution using the bump filter. The parameter h effectively controls the degree of the smoothing. A Programmable Filter for this is shown in Programmable Filter 2.

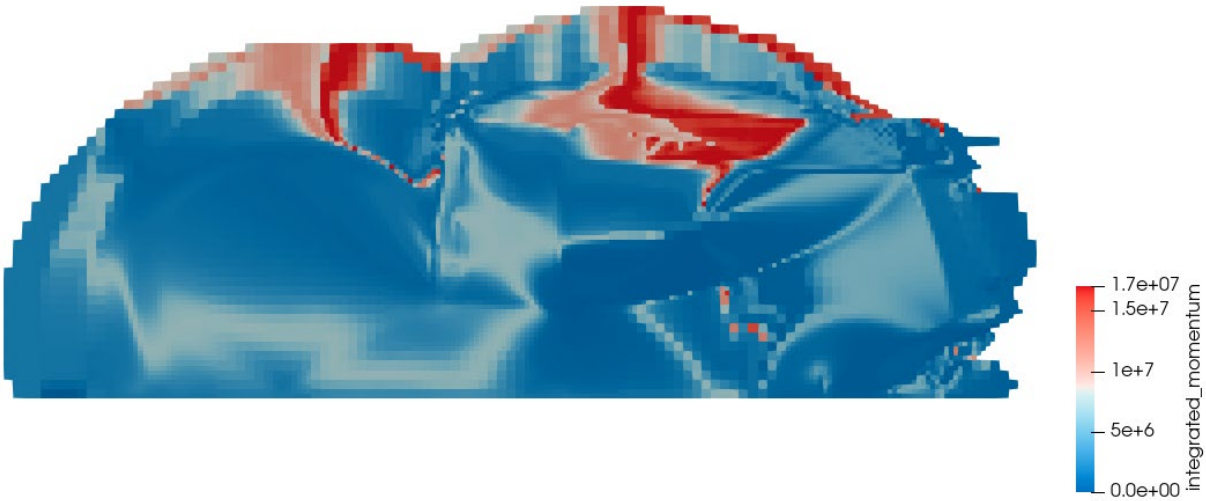

Figure 3 - Physical-space representation of the total momentum associated with a given polar angle in phase space.

Computing Maximum Momentum for a Given Polar Angle

It is useful to compute the maximum momentum at a given polar angle, particularly for scaling purposes. We will apply the same smoothing operation as before, with a different normalization.

The intent of this normalization is to preserve the actual maximum value where appropriate, while smoothing the effects of the discretization. To this end, we will simply normalize the bump function $\Psi(x,h)$ by its peak value, $\Psi(0,h)$. This yields:

$$\Psi'(x.h) \equiv e\,\Psi(x,h)$$

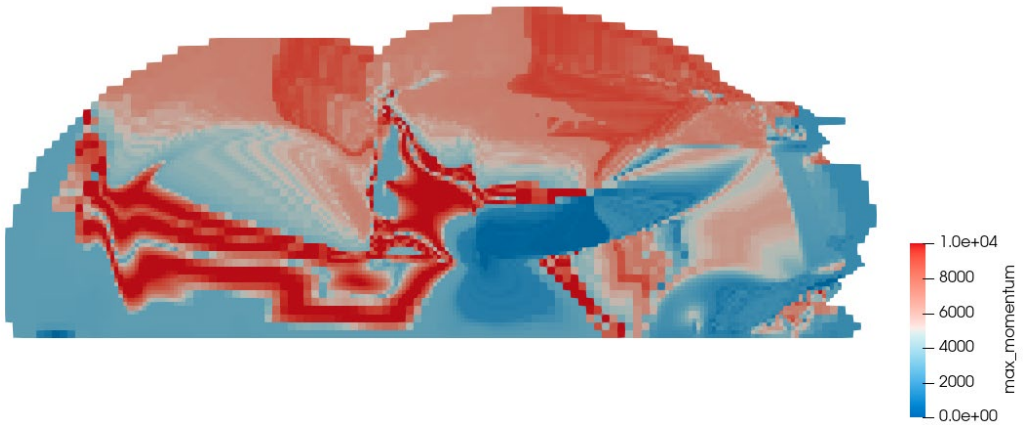

Figure 4 - Physical-space representation of the maximum momentum associated with a given phase-space polar angle.

Profiling has shown that the bulk of the computation time in these smoothing operations lies in the computation of $\Psi(x,h)$, so we take advantage of the work already done in computing the angular distribution by computing the maximum pressure at the same time, as shown in Programmable Filter 2

Representing the Angular Distribution as a Curve in Paraview

As computed above, both the `integrated_momentum` and `max_momentum` fields are cell-centered fields across the whole discrete mesh, representing the integrated and maximum momentum along the cone of constant polar angle in phase space, corresponding to the phase space angle of the momentum of the given cell. This is very confusing, and it is also not typically what we want. A better representation of the angular distribution of momentum would be a simple polar graph.

A simple polar graph can be created by using the integrated momentum as the radial distance from the origin and the phase space polar angle. A Programmable Filter to do this is shown in Programmable Filter 3. The resulting plot is shown in Figure 5.

```
p = inputs[0].CellData['cell_momentum']
q = inputs[0].CellData['theta']
# compute the magnitude of the momentum:
p = sqrt((p*p).sum(axis=1))
h = pi/32
h2 = h**2
sortargs = argsort(q)
unsortargs = argsort(sortargs)
q = q[sortargs]
p = p[sortargs]
P = 0*p
p_max = 0*p
for ind in range(len(q)):
  q0 = q[ind]
  min_ind, max_ind = searchsorted(q, (q0-h, q0+h))
  x2 = (q[min_ind:max_ind] - q0)**2
  Psi = exp(h2/(x2-h2))
  temp = Psi*p[min_ind:max_ind]
  P[ind] = sum(temp/(h*.443994))
  p_max[ind] = max(temp*2.71828)
output.CellData.append(P[unsortargs], 'integrated_momentum')
output.CellData.append(p_max[unsortargs], 'max_momentum')
```

Programmable Filter 2 - Code for computing the angular distribution of a vector quantity, as well as a smoothed maximum value. The smoothing function is applied to the polar angle directly.

```
P = inputs[0].CellData['integrated_momentum']
q = inputs[0].CellData['theta']
sortargs = argsort(q)
y = P*cos(q)
x = P*sin(q)
z = 0*P
x, y, z = (coord[sortargs] for coord in (x,y,z))
numPts = len(q)
pdo = self.GetPolyDataOutput()
newPts = vtk.vtkPoints()
for ind in range(0, numPts):
  newPts.InsertPoint(ind, x[ind], y[ind], z[ind])
pdo.SetPoints(newPts)
aPolyLine = vtk.vtkPolyLine()
aPolyLine.GetPointIds().SetNumberOfIds(numPts)
for ind in range(0, numPts):
  aPolyLine.GetPointIds().SetId(ind, ind)
pdo.Allocate(1, 1)
pdo.InsertNextCell(aPolyLine.GetCellType(), aPolyLine.GetPointIds())
```

Programmable Filter 3 - Code for creating a polar plot representation of the angular momentum distribution. The Output Data Set Type field must be set to vtkPolyData.

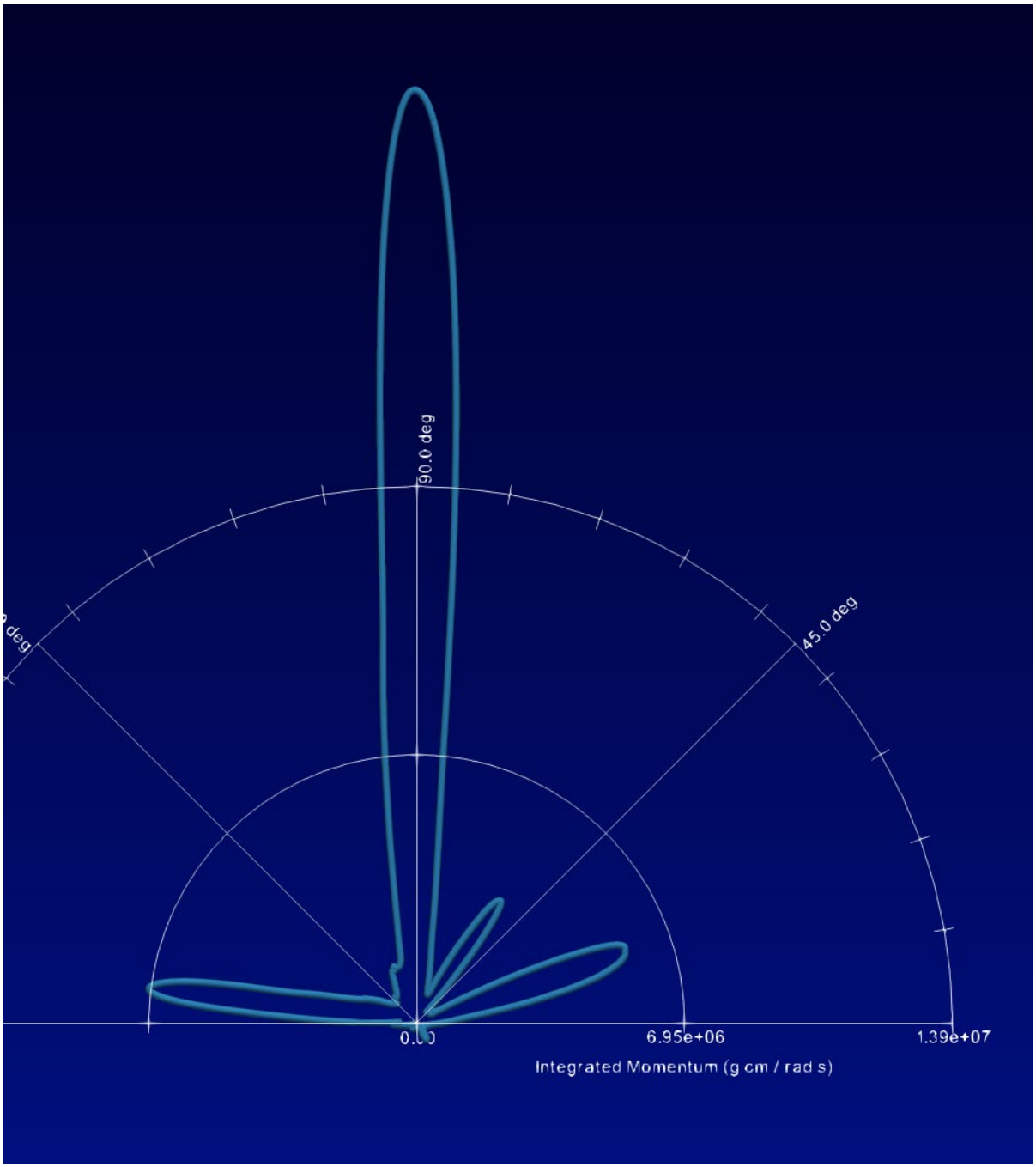

Figure 5 - Polar plot of angular momentum distribution for the shaped charge simulation.

Filling in the Angular Distribution

It is often desirable to add additional information to a given visualization. In this case, the most obvious way to do is by using Paraview's capabilities to fill in the curve we have drawn with a representation of the data on the original grid. This provides several challenges.

First, the curve we have drawn is rendered with respect to the phase space of the simulation. In other words, the x- and y- coordinates of the curve represent momentum values, not position. Paraview allows the user to convert a data set to an alternate coordinate representation using the Coordinate Values option in the Calculator filter. The question becomes, "What is the appropriate coordinate transformation?" One choice is to scale the cell momentum:

$$p_{scaled} \equiv p \frac{p_{integrated}}{p_{maximum}}$$

Having done this, it is possible to convert the rest of the grid values to the scaled momentum coordinates. It is then possible to plot a scalar field, such as the magnitude of the individual cell momentum, as shown in Figure 6.

Further Thoughts and Considerations

Visual Distortion Due to Spherical Coordinates

One problem with this analysis is that the volume over which momentum is integrated is smaller near the poles than it is near the equator. It is possible to modify the Programmable Filter code to integrate over something like $a \equiv \cos\theta$ in order to address this. However, care must be taken to ensure that the integration is done correctly, and this will result in an artificially symmetric result for $180^\circ < \theta < 360^\circ$. That may not matter for simulations that have run to late time, where momentum toward the axis of symmetry may be neglected.

How to Color the Fill Points

Coloring the fill points by cell momentum is simple, but it can also be misleading. The xRage quantity `cell_momentum` is extrinsic, which means that it will scale both with radial distance and with grid size. The dark red points in Figure 6, for instance, are simply representations of grid points that have been de-refined by the xRage AMR. It is likely that filling in the curve will require something more sophisticated.

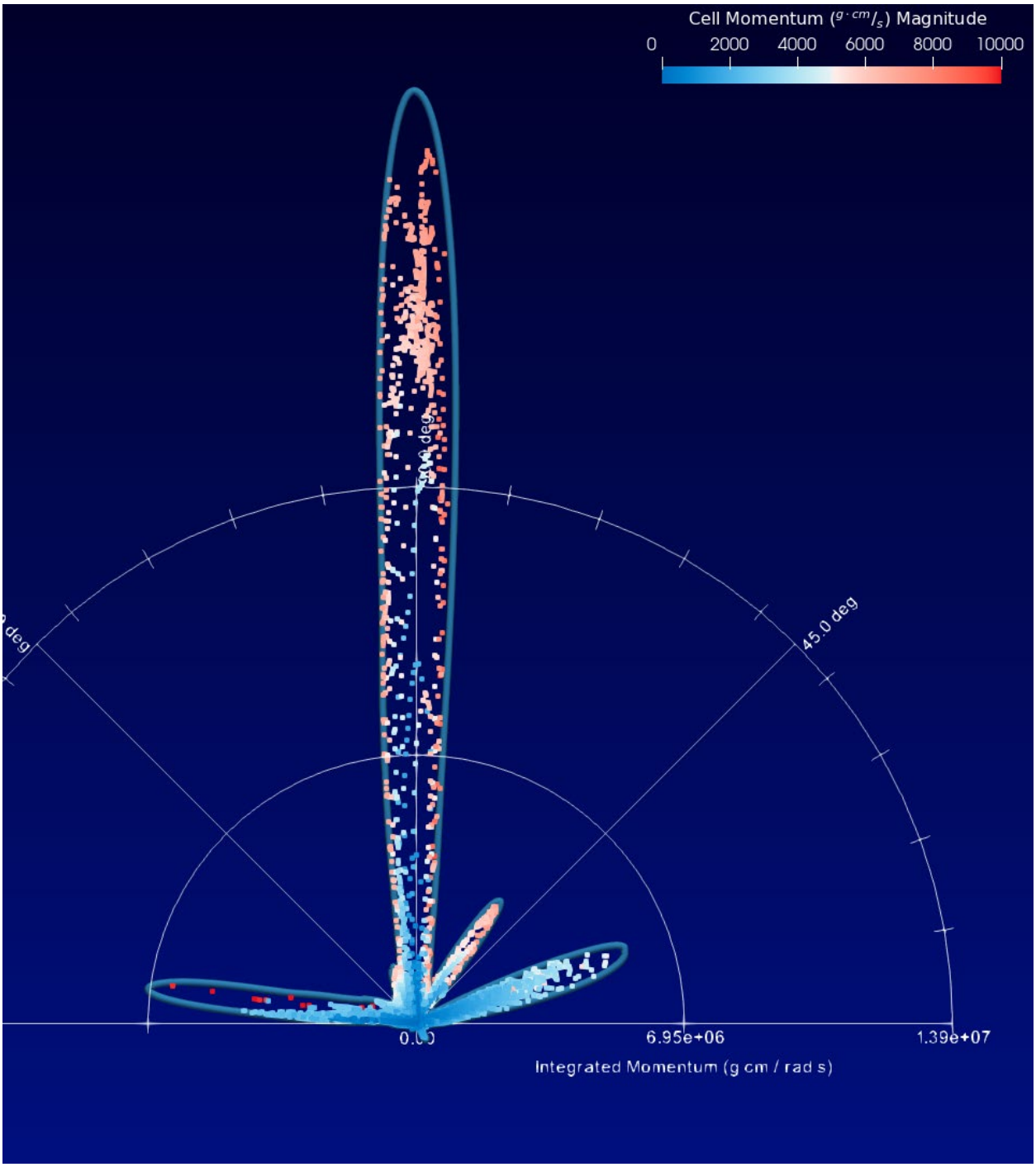

Figure 6 - Polar plot of angular momentum distribution, filled with pointwise representations of the cell momentum contained in each cell.

Python state file for creating this analysis

```
# state file generated using paraview version 5.9.1

#### import the simple module from the paraview
from paraview.simple import *
#### disable automatic camera reset on 'Show'
paraview.simple._DisableFirstRenderCameraReset()

# ----------------------------------------------------------------
# setup the data processing pipelines
# ----------------------------------------------------------------

# create a new 'PIO Reader'
shapedcharge2dpio = PIOReader(registrationName='shapedcharge2d.pio',
FileName='U:\\Visualization\\pio\\shapedcharge2d.pio')
shapedcharge2dpio.CellArrays = ['cell_energy', 'cell_momentum',
'mass', 'mat', 'vcell']

# create a new 'Calculator'
calculator1 = Calculator(registrationName='Calculator1',
Input=shapedcharge2dpio)
calculator1.AttributeType = 'Cell Data'
calculator1.ResultArrayName = 'cell_momentum'
calculator1.Function =
'cell_momentum_X*iHat+cell_momentum_Y*jHat+0*kHat'

# create a new 'Merge Blocks'
mergeBlocks1 = MergeBlocks(registrationName='MergeBlocks1',
Input=calculator1)

# create a new 'Ghost Cells Generator'
ghostCellsGenerator1 =
GhostCellsGenerator(registrationName='GhostCellsGenerator1',
Input=mergeBlocks1)
ghostCellsGenerator1.MinimumNumberOfGhostLevels = 2

# create a new 'Reflect'
reflect1 = Reflect(registrationName='Reflect1',
Input=ghostCellsGenerator1)
reflect1.CopyInput = 0

# create a new 'Transform'
transform1 = Transform(registrationName='Transform1', Input=reflect1)
transform1.Transform = 'Transform'

# init the 'Transform' selected for 'Transform'
transform1.Transform.Rotate = [0.0, 0.0, -90.0]

# create a new 'Calculator'
calculator2 = Calculator(registrationName='Calculator2',
Input=transform1)
calculator2.AttributeType = 'Cell Data'
calculator2.ResultArrayName = 'p_x_mag'
```

```
calculator2.Function = 'abs(cell_momentum_X)'

# create a new 'Threshold'
threshold1 = Threshold(registrationName='Threshold1',
Input=calculator2)
threshold1.Scalars = ['CELLS', 'p_x_mag']
threshold1.ThresholdRange = [1e-14, 1e+33]

# create a new 'Programmable Filter'
programmableFilter1 =
ProgrammableFilter(registrationName='ProgrammableFilter1',
Input=threshold1)
programmableFilter1.Script = """p =
inputs[0].CellData['cell_momentum']

def atan2(y,x):
  sgnx = sign(x)
  sgny = sign(y)
  # This fails when x = 0, b/c arctan(y/x) = NaN, and 0*NaN = NaN.
  # Make sure you've gotten rid of those using Threshold or similar.
  out = sgnx**2*arctan(y/x)+.5*(1-sgnx)*(1+sgny-sgny**2)*pi
  return out

output.CellData.append(atan2(p[:,1], p[:,0]), 'theta')"""
programmableFilter1.RequestInformationScript = ''
programmableFilter1.RequestUpdateExtentScript = ''
programmableFilter1.CopyArrays = 1
programmableFilter1.PythonPath = ''

# create a new 'Calculator'
calculator5 = Calculator(registrationName='Calculator5',
Input=programmableFilter1)
calculator5.AttributeType = 'Cell Data'
calculator5.ResultArrayName = 'Polar Angle (deg)'
calculator5.Function = 'theta*180/3.14159'

# create a new 'Programmable Filter'
programmableFilter2 =
ProgrammableFilter(registrationName='ProgrammableFilter2',
Input=programmableFilter1)
programmableFilter2.Script = """p =
inputs[0].CellData['cell_momentum']
q = inputs[0].CellData['theta']

# compute the magnitude of the momentum:
p = sqrt((p*p).sum(axis=1))
h = pi/32
h2 = h**2
sortargs = argsort(q)
unsortargs = argsort(sortargs)
q = q[sortargs]
p = p[sortargs]
P = 0*p
p_max = 0*p
```

```
for ind in range(len(q)):
  q0 = q[ind]
  min_ind, max_ind = searchsorted(q, (q0-h, q0+h))
  x2 = (q[min_ind:max_ind] - q0)**2
  Psi = exp(h2/(x2-h2))
  temp = Psi*p[min_ind:max_ind]
  P[ind] = sum(temp/(h*.443994))
  p_max[ind] = max(temp*2.71828)
output.CellData.append(P[unsortargs], 'integrated_momentum')
output.CellData.append(p_max[unsortargs], 'max_momentum')"""
programmableFilter2.RequestInformationScript = ''
programmableFilter2.RequestUpdateExtentScript = ''
programmableFilter2.CopyArrays = 1
programmableFilter2.PythonPath = ''

# create a new 'Calculator'
calculator3 = Calculator(registrationName='Calculator3',
Input=programmableFilter2)
calculator3.AttributeType = 'Cell Data'
calculator3.ResultArrayName = 'scaled_p_coords'
calculator3.Function =
'cell_momentum*integrated_momentum/max_momentum'

# create a new 'Cell Data to Point Data'
cellDatatoPointData1 =
CellDatatoPointData(registrationName='CellDatatoPointData1',
Input=calculator3)
cellDatatoPointData1.ProcessAllArrays = 0
cellDatatoPointData1.CellDataArraytoprocess = ['scaled_p_coords']
cellDatatoPointData1.PassCellData = 1

# create a new 'Calculator'
calculator4 = Calculator(registrationName='Calculator4',
Input=cellDatatoPointData1)
calculator4.CoordinateResults = 1
calculator4.Function = 'scaled_p_coords'

# create a new 'Calculator'
calculator6 = Calculator(registrationName='Calculator6',
Input=calculator4)
calculator6.AttributeType = 'Cell Data'
calculator6.ResultArrayName = 'Velocity (cm/s)'
calculator6.Function = 'cell_momentum/mass'

# create a new 'Threshold'
threshold2 = Threshold(registrationName='Threshold2',
Input=calculator2)
threshold2.Scalars = ['CELLS', 'mat']
threshold2.ThresholdRange = [1.5, 10630.557989487197]

# create a new 'Extract Time Steps'
extractTimeSteps1 =
ExtractTimeSteps(registrationName='ExtractTimeSteps1',
Input=threshold2)
```

```
extractTimeSteps1.TimeStepIndices = [0]
extractTimeSteps1.TimeStepRange = [0, 33]

# create a new 'Programmable Filter'
programmableFilter3 =
ProgrammableFilter(registrationName='ProgrammableFilter3',
Input=programmableFilter2)
programmableFilter3.OutputDataSetType = 'vtkPolyData'
programmableFilter3.Script = """P =
inputs[0].CellData['integrated_momentum']
q = inputs[0].CellData['theta']
sortargs = argsort(q)

x = P*cos(q)
y = P*sin(q)
z = 0*P

x, y, z = (coord[sortargs] for coord in (x,y,z))

numPts = len(q)
pdo = self.GetPolyDataOutput()
newPts = vtk.vtkPoints()
for ind in range(0, numPts):
  newPts.InsertPoint(ind, x[ind], y[ind], z[ind])
newPts.InsertPoint(numPts, x[0], y[0], z[0])
pdo.SetPoints(newPts)
aPolyLine = vtk.vtkPolyLine()
aPolyLine.GetPointIds().SetNumberOfIds(numPts+1)
for ind in range(0, numPts+1):
  aPolyLine.GetPointIds().SetId(ind, ind)
pdo.Allocate(1, 1)
pdo.InsertNextCell(aPolyLine.GetCellType(),
aPolyLine.GetPointIds())"""
programmableFilter3.RequestInformationScript = ''
programmableFilter3.RequestUpdateExtentScript = ''
programmableFilter3.PythonPath = ''

# create a new 'Tube'
tube1 = Tube(registrationName='Tube1', Input=programmableFilter3)
tube1.Scalars = ['POINTS', '']
tube1.Vectors = ['POINTS', '1']
tube1.Radius = 100000.0
```